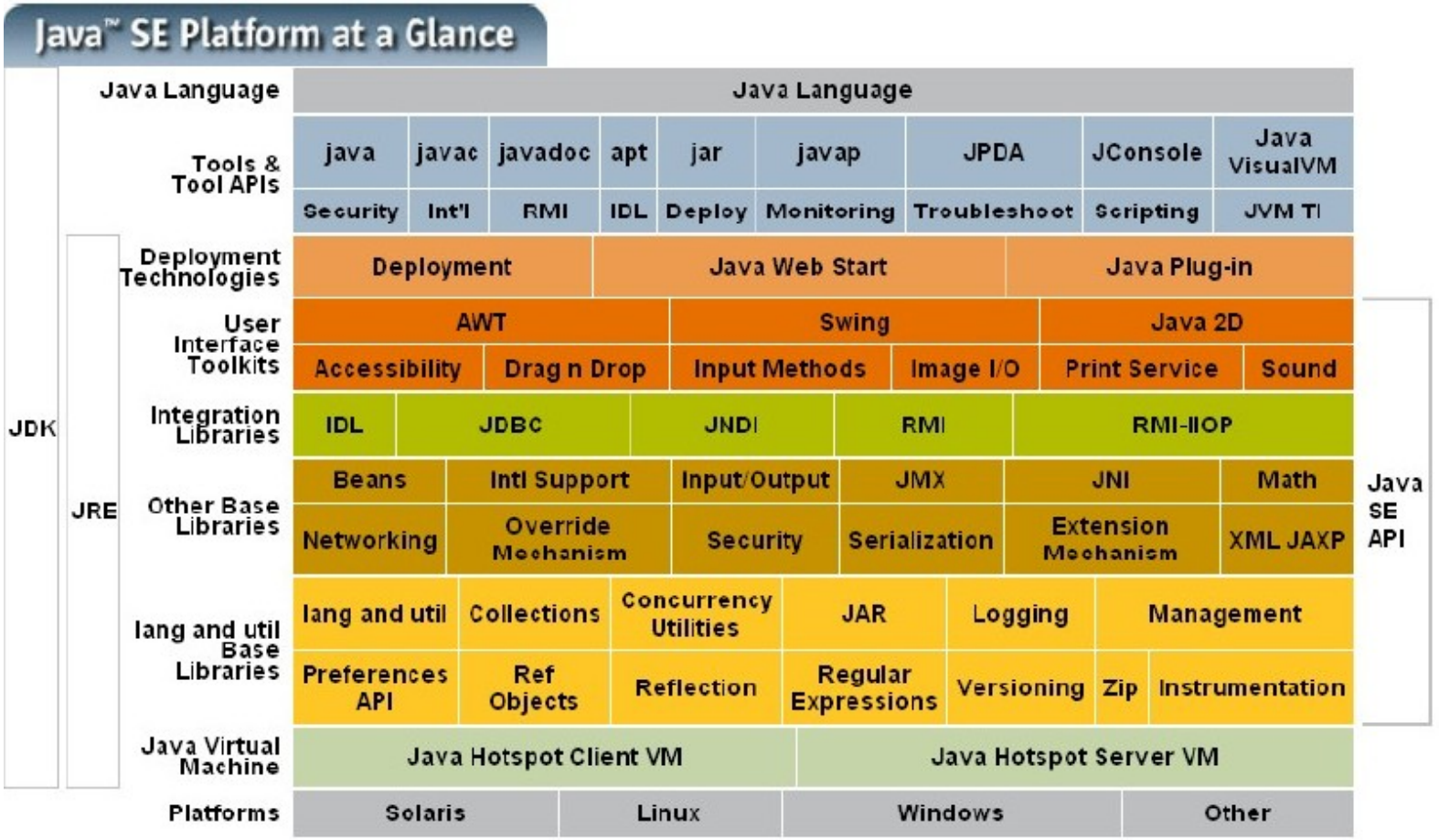
JVM内存管理机制及其调优

关心

你遇到过什么问题？

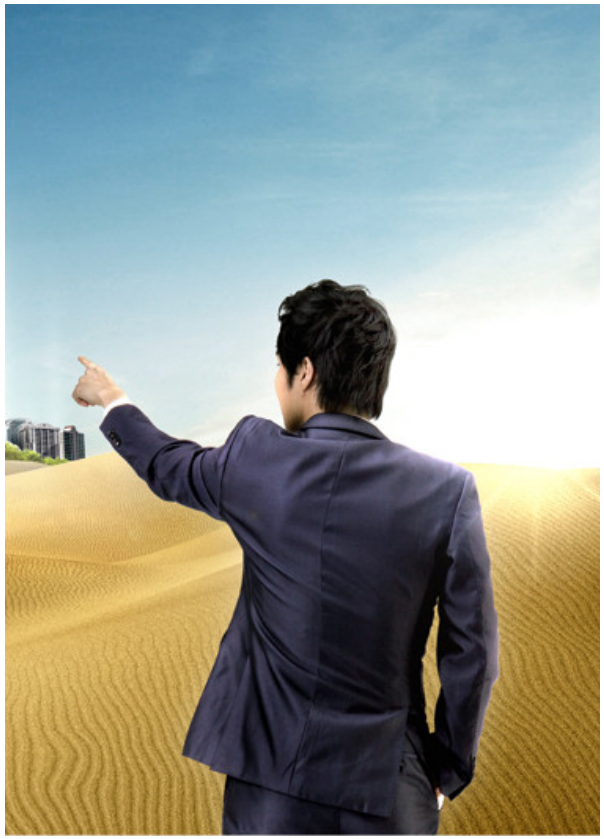
- OOM: Heap, Stack, Perm
- 系统频繁GC
- Java进程占用CPU过高
- Java占用内存增长很快
- 远程调用timeout
- 系统响应时间变长，越来越慢
- ……

JAVA架构图


Java™ SE Platform at a Glance
JDK
JRE
Java SE API
Java Language
Java Language
Tools & Tool APIs
java
javac
javadoc
apt
jar
javap
JPDA
JConsole
Java VisualVM
Security
Int'l
RMI
IDL
Deploy
Monitoring
Troubleshoot
Scripting
JVM TI
Deployment Technologies
Deployment
Java Web Start
Java Plug-in
User Interface Toolkits
AWT
Swing
Java 2D
Accessibility
Drag n Drop
Input Methods
Image I/O
Print Service
Sound
Integration Libraries
IDL
JDBC
JNDI
RMI
RMI-IIOP
Other Base Libraries
Beans
Intl Support
Input/Output
JMX
JNI
Math
Networking
Override Mechanism
Security
Serialization
Extension Mechanism
XML JAXP
lang and util Base Libraries
lang and util
Collections
Concurrency Utilities
JAR
Logging
Management
Preferences API
Ref Objects
Reflection
Regular Expressions
Versioning
Zip
Instrumentation
Java Virtual Machine
Java Hotspot Client VM
Java Hotspot Server VM
Platforms
Solaris
Linux
Windows
Other

导航

- 1.什么是JVM
- 2.JVM内存模型及内存分配
- 3.GC垃圾回收
- 4.JVM监控工具
- 5.调优



JVM是Sun发布的一种规范

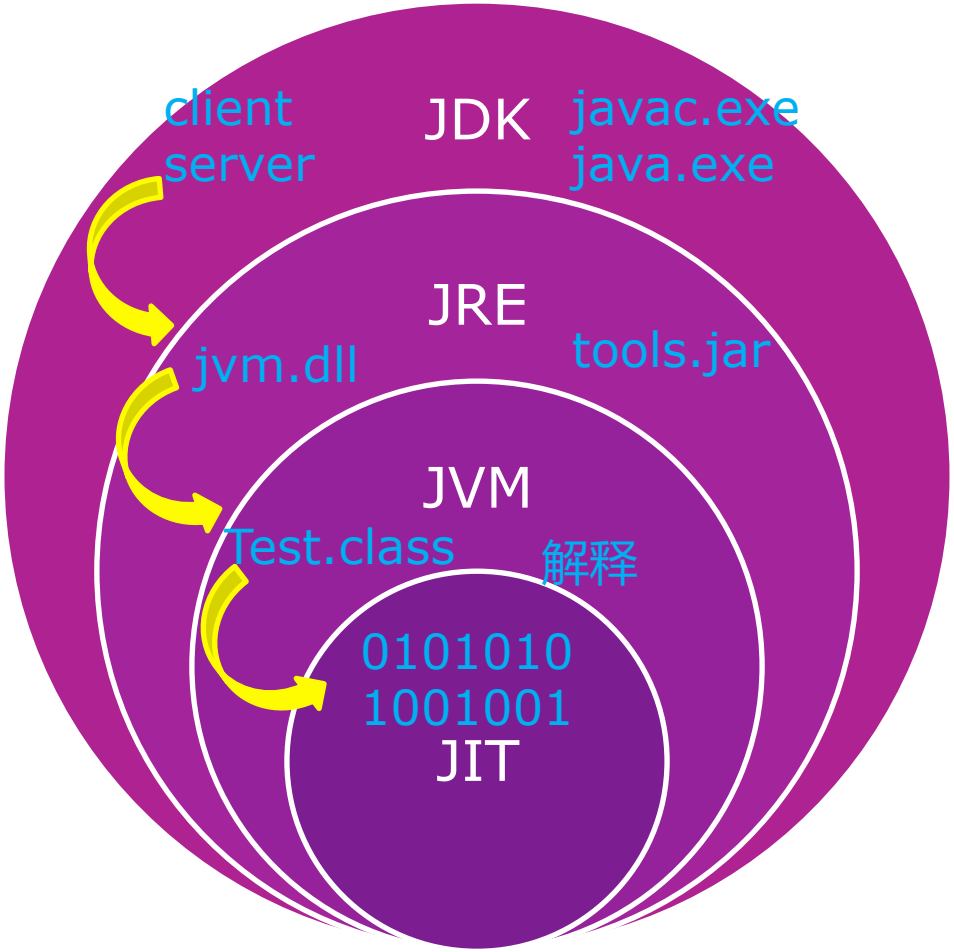
- 基本上来说，JVM是一个虚拟运行环境，对于字节码来说就像是一个机器一样，可以执行任务，并通过底层实现执行内存相关的操作。
- JVM使Java程序做到了“一次编写，到处运行”。
- JVM解放了程序员，使程序员不必再关系对象的生命周期，使程序员不必再关心应该在何时释放内存。
- 可以将JVM当做是一种专为Java而生的特殊的操作系统，它的工作是管理运行Java应用程序的运行时环境。

JVM的实现产品

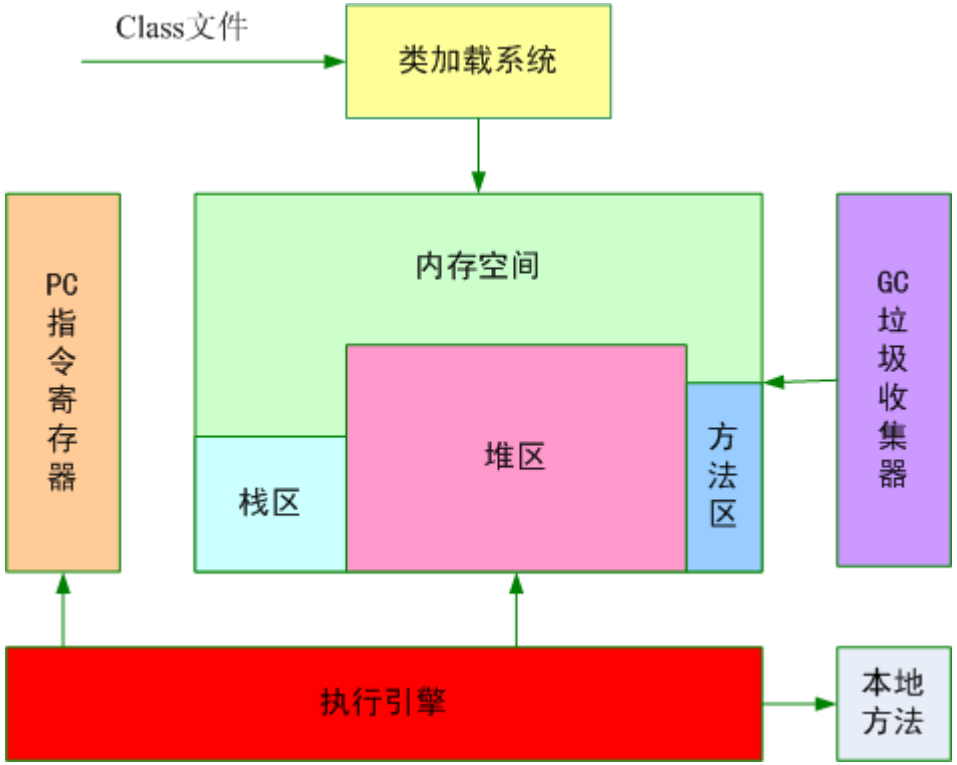
- CEE-J
- Excelsior JET
- Hewlett-Packard
- J9 (IBM)
- Jbed
- Jblend
- Jrockit
- MRJ
- MicroJvm
- MS JVM
- OJVM
- PERC
- Blackdown Java
- CVM
- Gemstone
- Golden Code Development
- Intent
- Novell
- NSIcom CrE-ME
- ChaiVM
- HotSpot
- AegisVM
- Apache Harmony
- CACAO
- Dalvik
- IcedTea
- IKVM.NET
- Jamiga
- JamVM
- Jaos
- JC
- Jelatine JVM
- JESSICA
- Jikes RVM
- Jnode
- JOP
- Juice
- Jupiter
- JX
- Kaffe
- leJOS
- Mika VM
- Mysaifu
- NanoVM
- SableVM
- Squawk virtual machine
- SuperWaba
- TinyVM
- VMkit of Low Level Virtual Machine
- Wonka VM
- Xam

EJC(Eclipse Java Compiler)

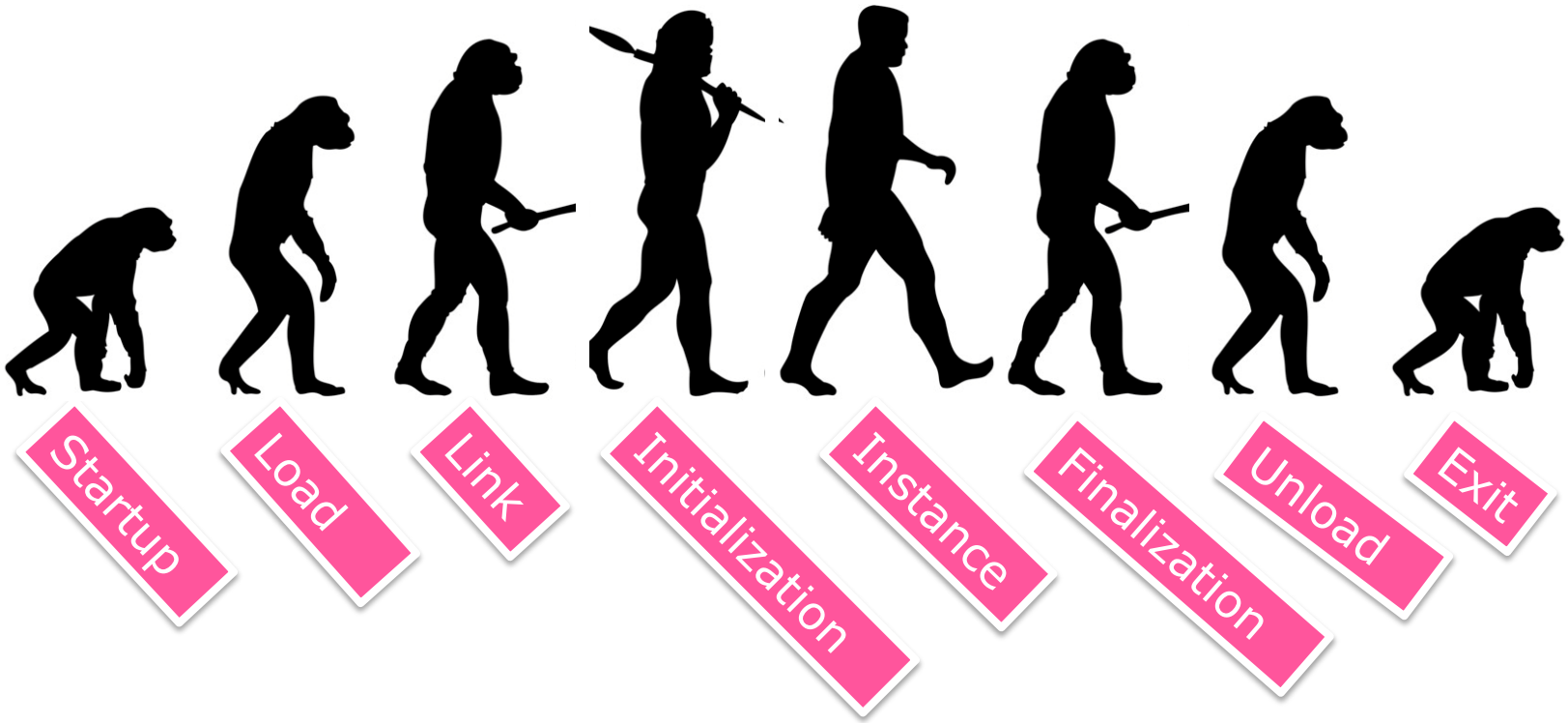
HotSpot JVM


client
server
JDK
javac.exe
java.exe
JRE
jvm.dll
tools.jar
JVM
Test.class
解释
0101010
1001001
JIT

JVM标准结构



JVM从生到死


Startup
Load
Link
Initialization
Instance
Finalization
Unload
Exit

导航

- 1.什么是JVM
- 2.JVM内存模型及内存分配
- 3.GC垃圾回收
- 4.JVM监控工具
- 5.调优

各个年代

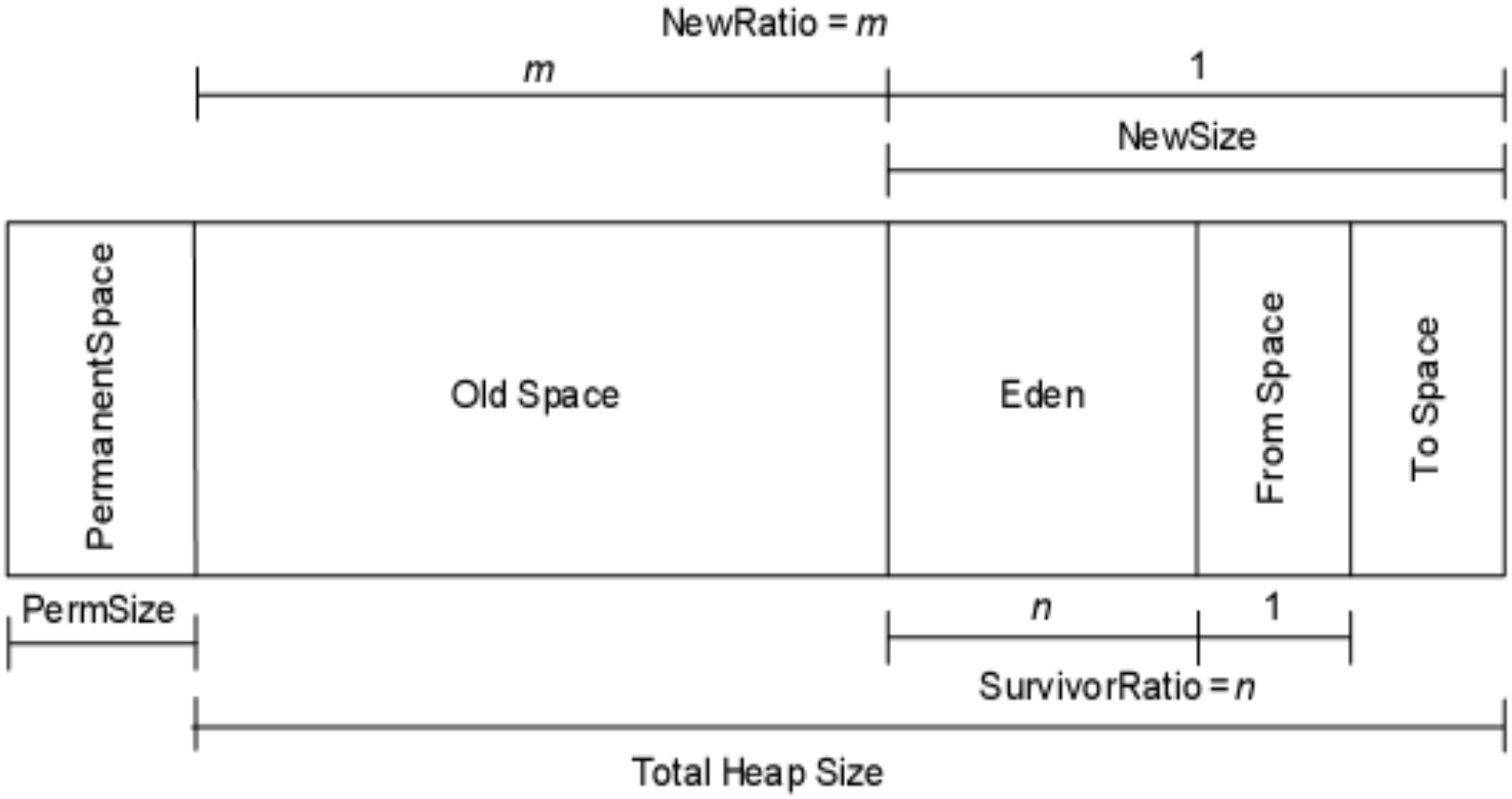
- 新生代（Young Generation）
  - 伊甸园空间（Eden ）、幸存者空间（Survivor ）

  最新被创建的对象会被分配到这里，由于大部分对象在创建后会很快变得不可到达，所以很多对象被创建在新生代，然后消失。
- 老年代（Old Generation）

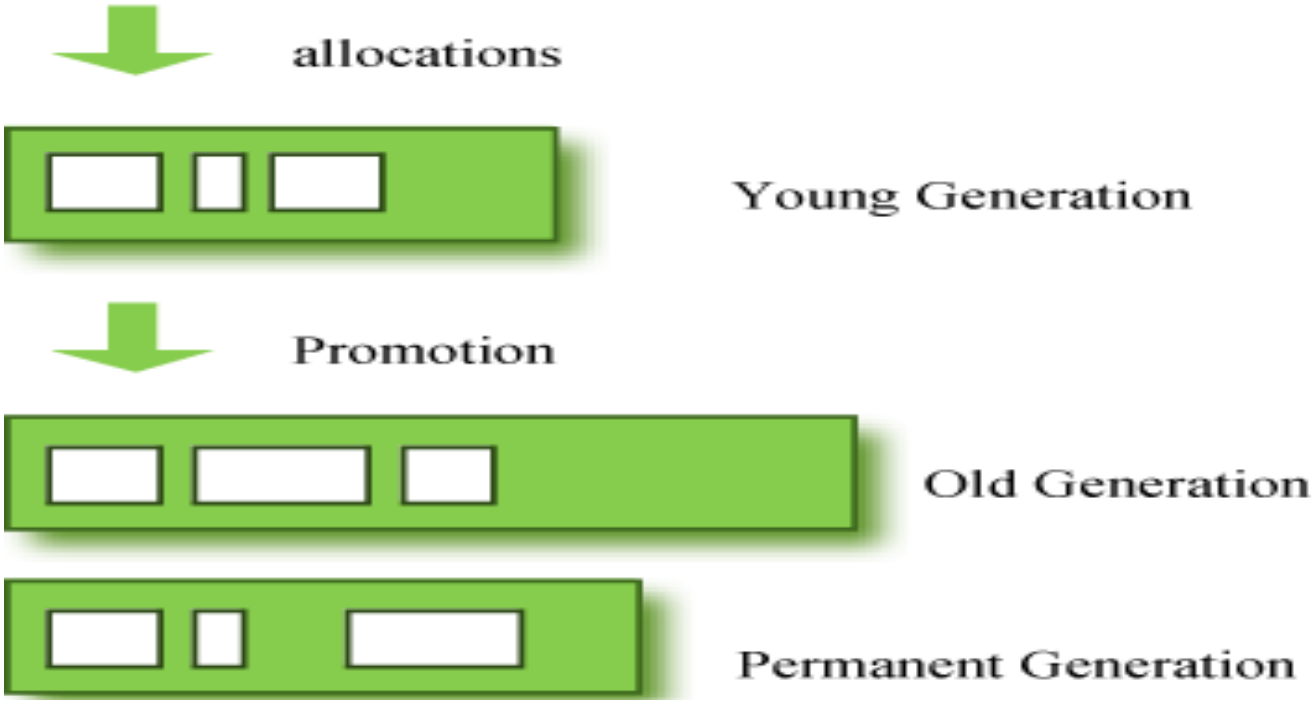
  对象没有变得不可达，并且从新生代中存活下来，会被拷贝到这里。
- 持久代（Permanent Generation）

  也被称为方法区（method area）。他用来保存类常量以及字符串常量

分代的汇总图


NewRatio = m
m
1
NewSize
PermanentSpace
Old Space
Eden
From Space
To Space
PermSize
n
1
SurvivorRatio = n
Total Heap Size

每个空间的执行顺序


allocations
Young Generation
Promotion
Old Generation
Permanent Generation

内存不能滥用

回收也需要耗费资源

用不好会无法回收

小技巧

- 测试机器最大可用内存

```
D:\jdk1.6.0_10\bin>java -Xmx2048m -version
Error occurred during initialization of VM
Could not reserve enough space for object heap
Could not create the Java virtual machine.
```

Heap堆

-Xms(默认内存1/64&小于1GB)

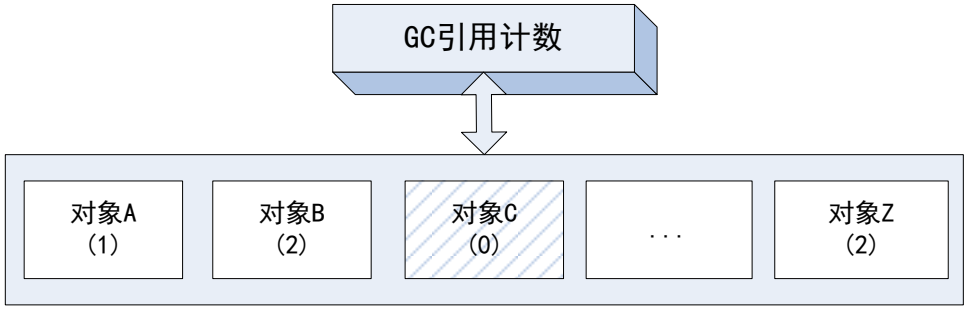
-Xmx(默认内存1/4&小于1GB)

导航

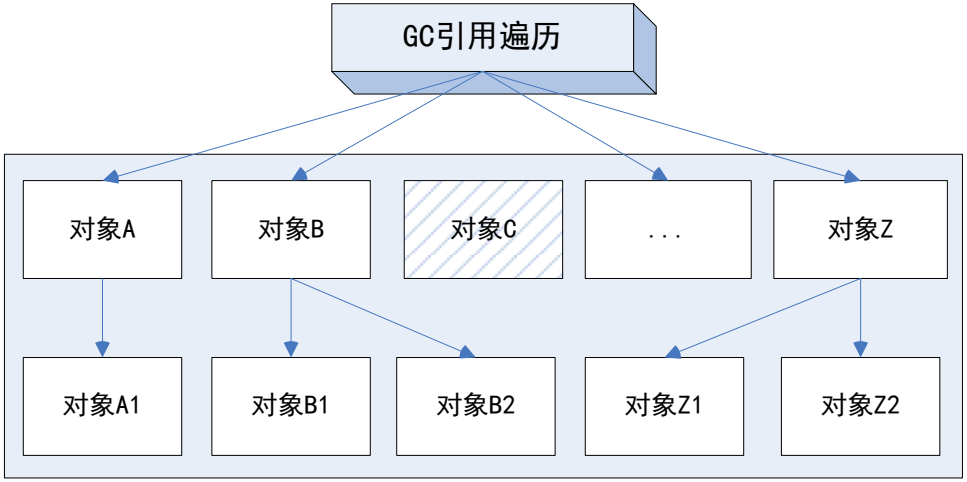
- 1.什么是JVM
- 2.JVM内存模型及内存分配
- 4.GC垃圾回收
- 5.监控及配置
- 6.JVM调优

对象何时回收

- 引用计数



- 对象引用遍历

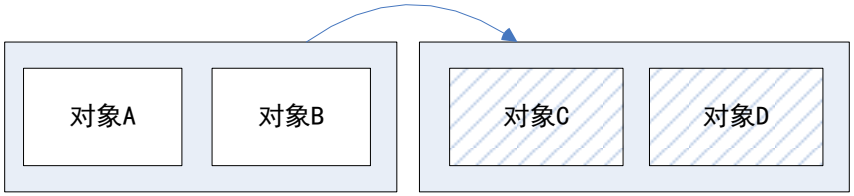


垃圾搜集算法

- 复制
- 标记清除
- 标记压缩

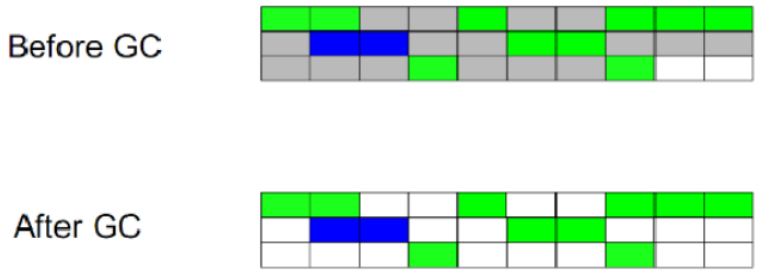
垃圾搜集算法1

- 复制
  - 找到活动对象拷贝到新的空间
  - 适合存活对象较少情况，增加内存成本高



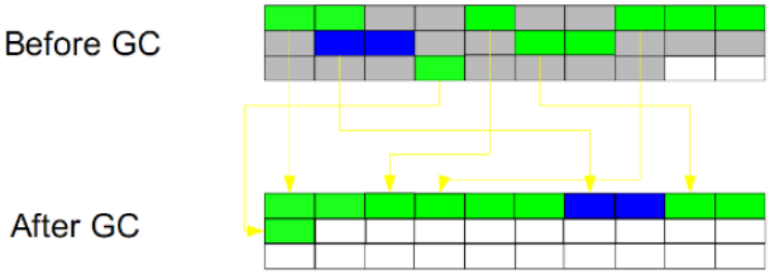
垃圾搜集算法2

- 标记清除
  - 从跟开始将活动对象标记，然后再扫描未标记的一次回收
  - 不需要移动对象，仅对不存活对象处理，适合存活对象较多情况，会造成内存碎片

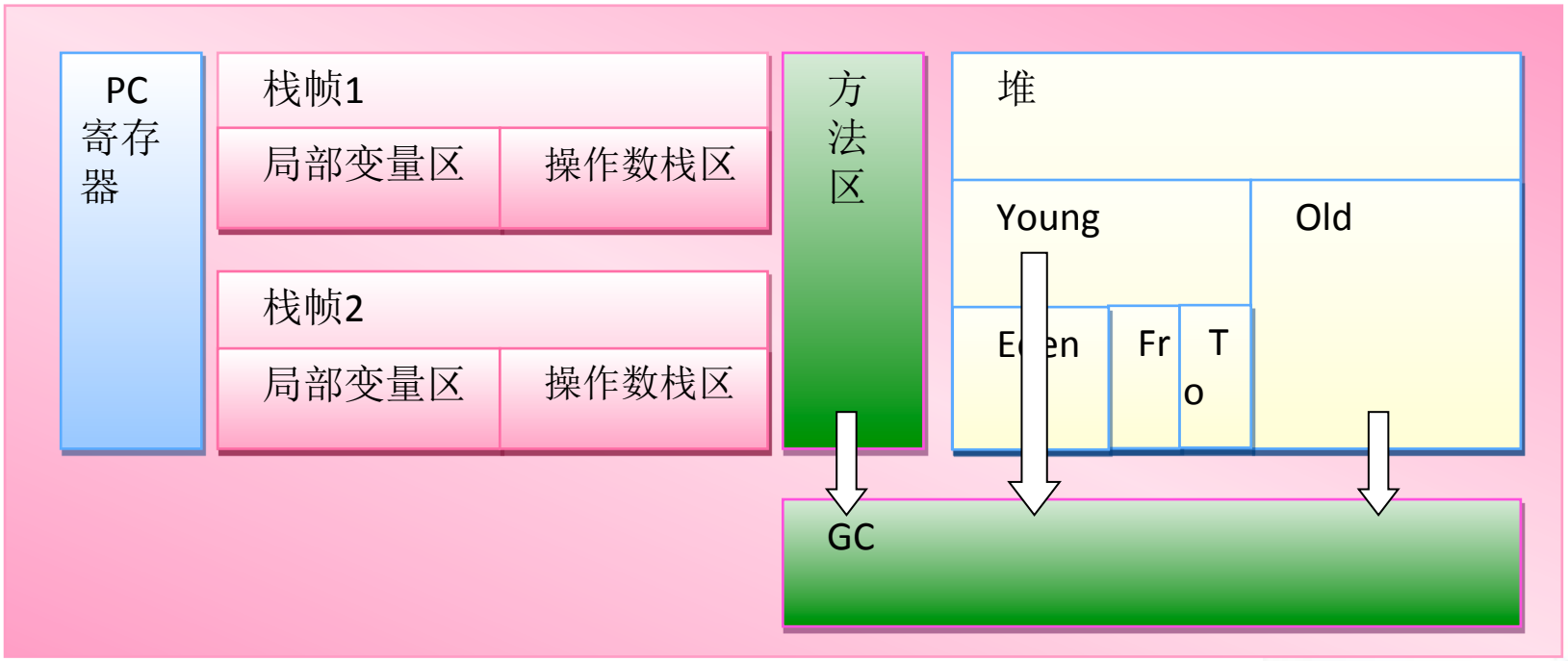


垃圾搜集算法3

- 标记压缩
  - 在标记清除基础上，往左移动存活对象
  - 成本高，好处是没有碎片



GC收集哪里的垃圾？



垃圾收集器类型

分类	垃圾收集器类型
线程数	串行
	并行
工作模式	并发
	独占
碎片处理	压缩
	非压缩
分代	新生代
	老年代

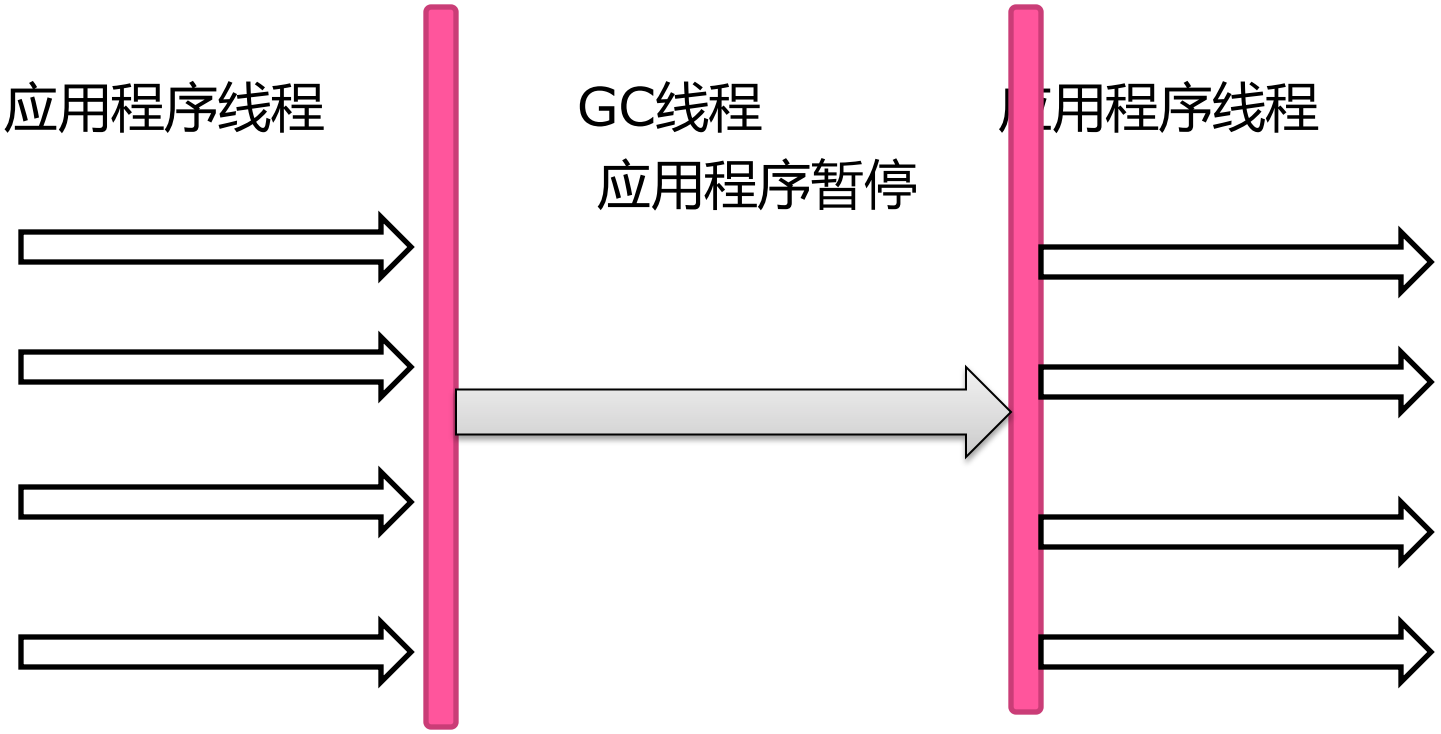
评价GC策略指标

- 吞吐量：
  - 吞吐量=GC消耗时间/应用运行总时间
  - 默认是-XX:GCTimeRatio=99%（程序运行100分，GC1分钟）

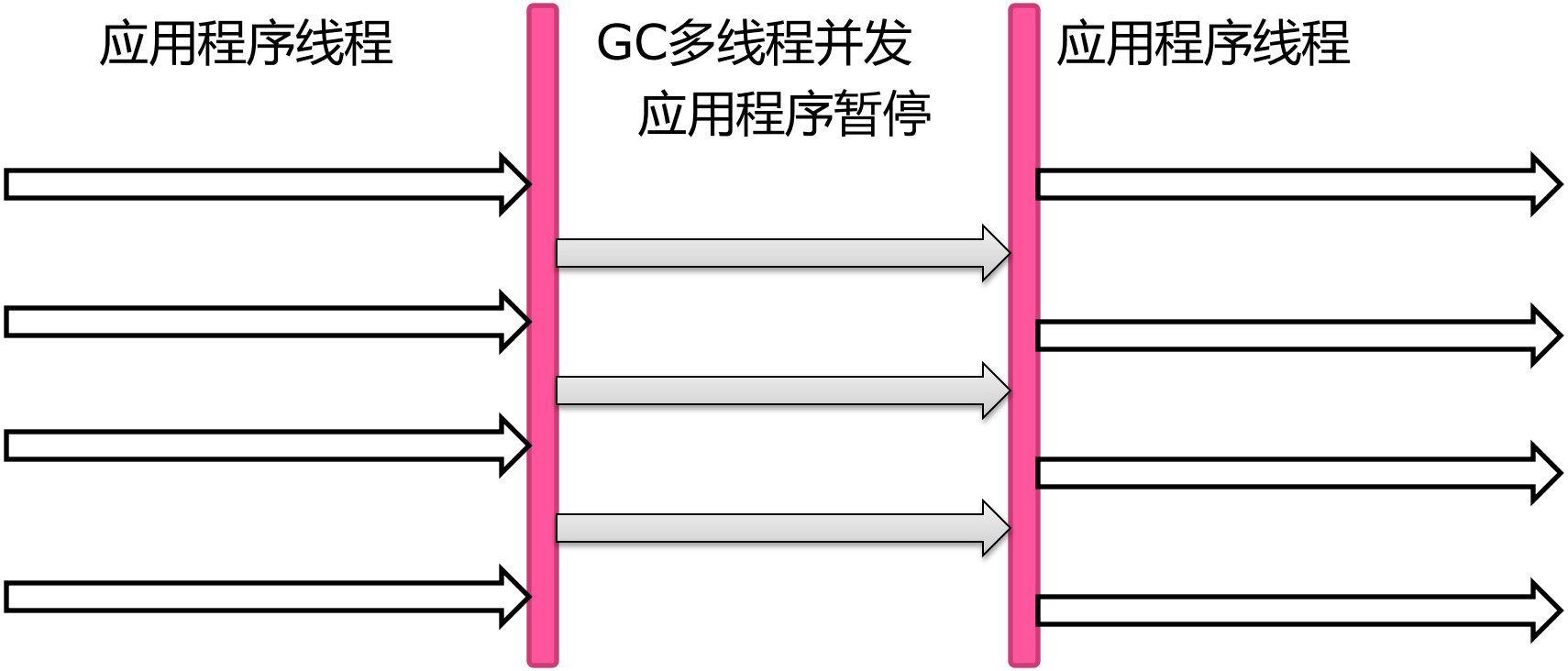
- 停顿时间：
  - 停顿时间=GC每次造成的应用暂停时间。
  - 默认不启用该策略。
  - 可通过-XX:MaxGCPauseMillis=n来设定停顿时间范围。

  - 如果以上两个参数都设定，则先满足暂停时间策略，再满足吞吐量策略。但总的目标应该是，降低GC对应用的影响。

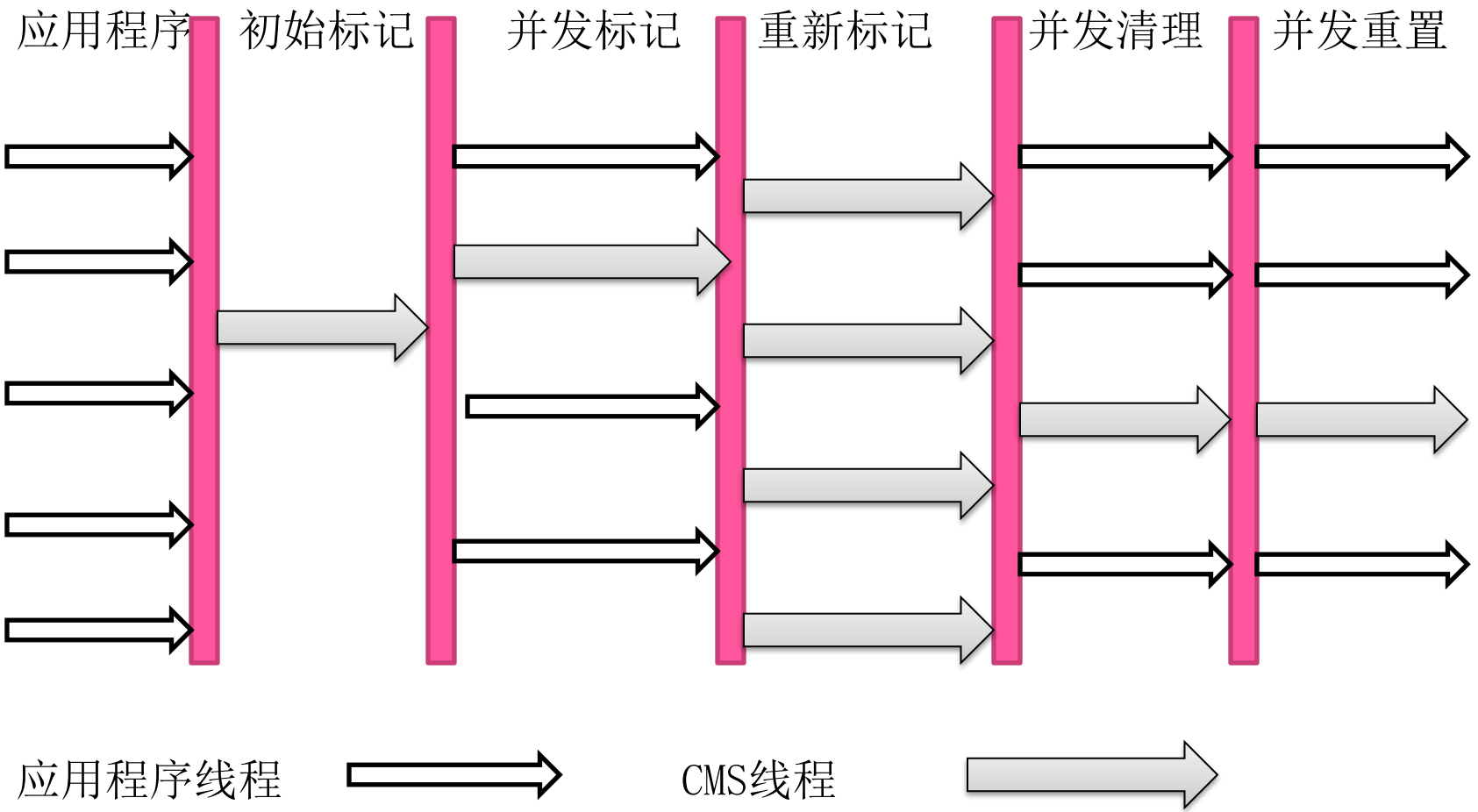
串行收集器



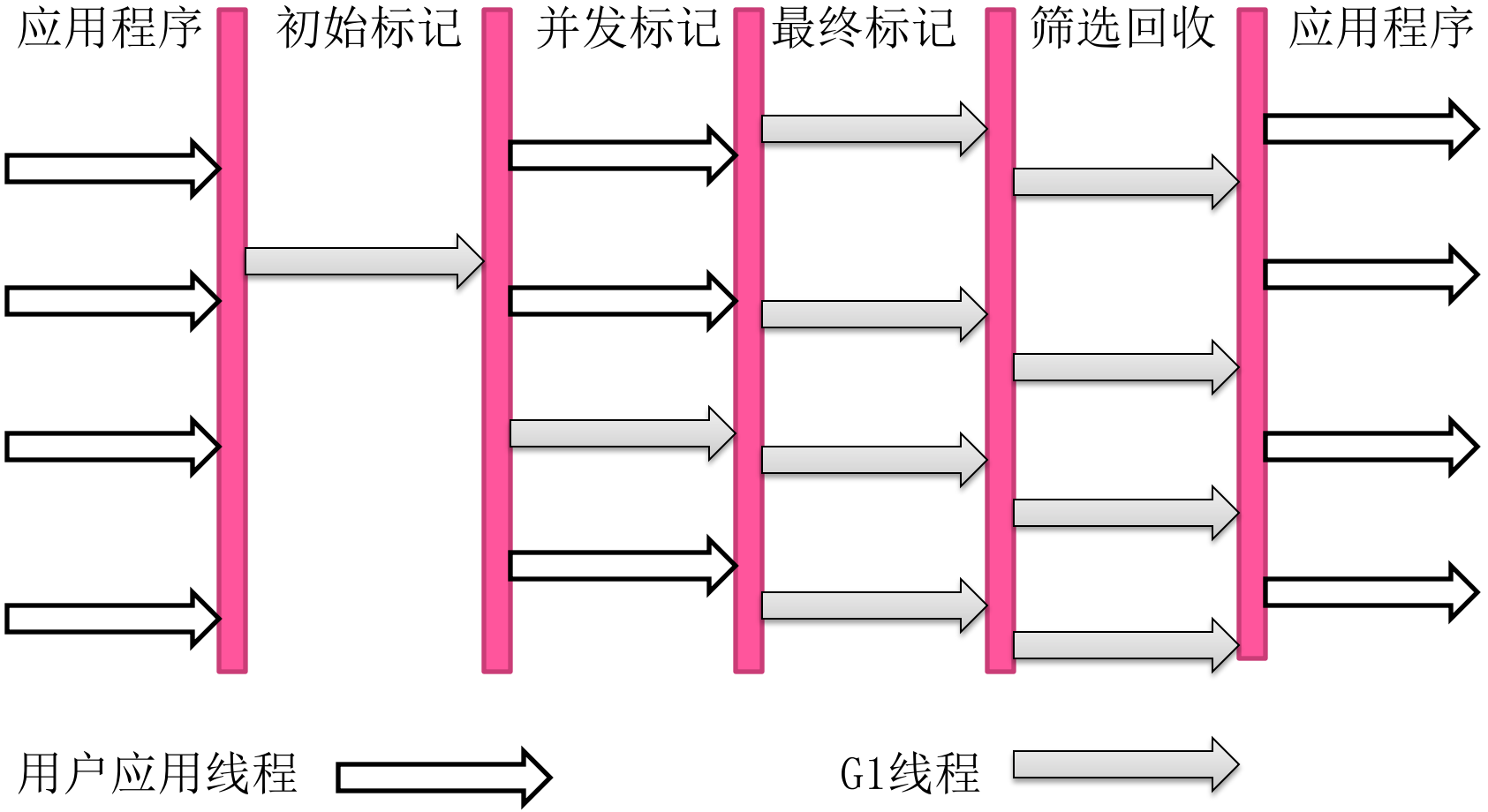
并行收集器



CMS收集器

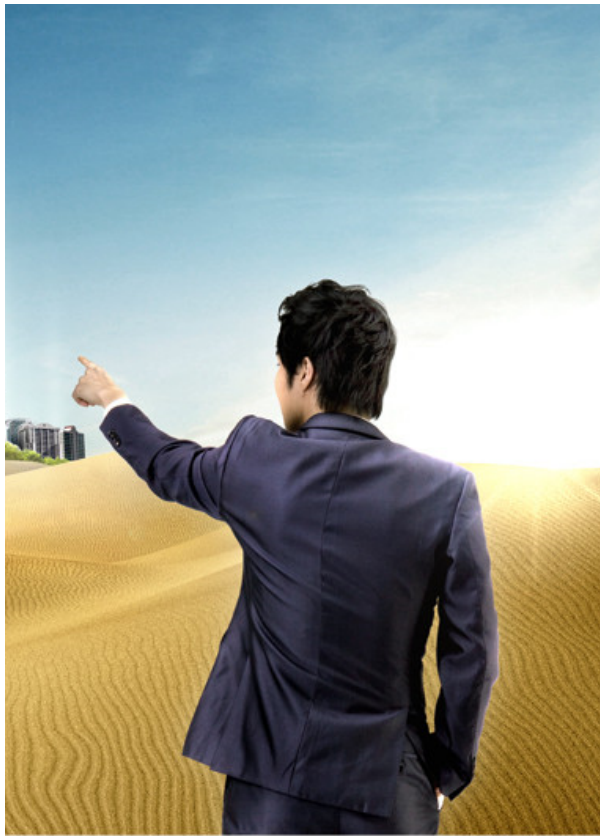


G1收集器



导航

- 1.什么是JVM
- 2.JVM内存模型及内存分配
- 3.GC垃圾回收
- 4.JVM监控工具
- 5.JVM调优



输出GC日志

- 输出到控制台
  - -XX:+PrintGC简要信息
  - -XX:+PrintGCDetails详细信
  - -XX:+PrintGCTimeStamps时间戳
  - -XX:+PrintGCApplicationStoppedTime暂停时间

```
java -Xms20M -Xmx20M -Xmn10M -XX:SurvivorRatio=8 -XX:+UseParallelGC -
verbose:GC -XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:
+PrintGCApplicationStoppedTime DemoPSGC
```

- 输出到文件
  - -Xloggc:/opt/gc.log

```
java -Xms20M -Xmx20M -Xmn10M -XX:SurvivorRatio=8 -XX:+UseParallelGC
-verbose:GC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -Xloggc:/opt/
test/gc.log DemoPSGC
```

JVM监控工具

- 命令行：
  - jstat（关注GC的次数和GC所耗时间）
  - jmap（将内存堆印象保存为文件）
  - HeapAnalyzer（分析jmap保存的文件）
- 图形化
  - Jconsole
  - VisualVM

JVM监控工具示例

jstat -gc $vmid$ 1000 （vmid是java应用的进程id号，1000：每隔一秒展示GC监控数据）

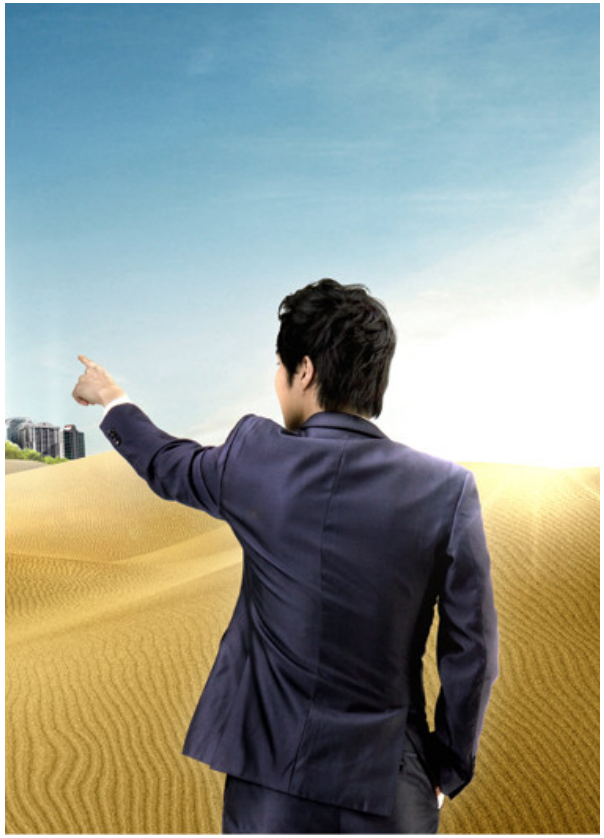
```
C:\Users\Administrator>jstat -gc 6816 1000
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       PC     PU    YGC     YGCT    FGC    FGCT     GCT
2368.0 2432.0  0.0    960.0  1567936.0 1039773.5 2621440.0   170334.7  256000.0 94587.7   1989   41.001   1      0.971   41.973
2368.0 2432.0  0.0    960.0  1567936.0 1059569.8 2621440.0   170334.7  256000.0 94587.7   1989   41.001   1      0.971   41.973
2368.0 2432.0  0.0    960.0  1567936.0 1145679.6 2621440.0   170334.7  256000.0 94587.7   1989   41.001   1      0.971   41.973
2368.0 2432.0  0.0    960.0  1567936.0 1188432.4 2621440.0   170334.7  256000.0 94587.7   1989   41.001   1      0.971   41.973
2368.0 2432.0  0.0    960.0  1567936.0 1222862.5 2621440.0   170334.7  256000.0 94587.7   1989   41.001   1      0.971   41.973
2368.0 2432.0  0.0    960.0  1567936.0 1246960.8 2621440.0   170334.7  256000.0 94587.7   1989   41.001   1      0.971   41.973
2368.0 2432.0  0.0    960.0  1567936.0 1271406.4 2621440.0   170334.7  256000.0 94587.7   1989   41.001   1      0.971   41.973
```

从上图中可计算得到以下值：
年青代GC发生了1989次（YGC），共耗费了41.001秒（YGCT），故年青代每次GC的时间为： 41.001/ 1989= 0.020秒
完全GC发生了1次（FGC），共耗费了0.971秒（FGCT），故完全GC每次GC的时间为：0.971/1=0.971秒

一般性能较好的判断：年青代GC耗时在50~100毫秒以内，完全GC小于1秒以内为很好的性能。（因为GC均会影响系统访问）

导航

- 1.什么是JVM
- 2.JVM内存模型及内存分配
- 3.CG垃圾回收
- 4.JVM监测工具
- 5.调优



调优前要做到

- 调查系统现状/寻找性能瓶颈/定位问题
  - 请求次数
  - 响应时间
  - 资源消耗：CPU/内存/文件/网络
- 确定调优目标
  - 单机用户量=用户量/机器数
  - 并发目标：如95%用户500ms响应
  - …

调优过程

- 定位资源消耗
  - 在哪里
    - CPU/内存/文件/网络
  - 什么类型
    - 线程调度太频繁
    - 线程load太高
    - Full GC太频繁
    - Full GC时间过长
    - 文件读写太慢、堵塞
- 定位问题代码
  - 使用Java工具分析内存、线程
  - 优化问题代码
  - 未来：编写高效代码

JVM调优在哪些方面

- 内存管理
  - 代大小配置：决定了YoungGC和FullGC
- GC策略
  - GC次数
  - GC时间
  - 应用暂停时间
- 程序优化

代大小调优策略

- 关键参数
  - 决定Heap大小：-xms -xmx –xmn(-xms=-xmx)
    - 取决与操作系统位数和CPU能力
  - Eden/From/To：决定YounGC：-XX:SurvivorRatio
  - 新生代存活周期：决定FullGC ：–XX:MaxTenuringThreshold
- 新生代/旧生代
  - 避免新生代设置过小
    - 频繁YoungGC
    - 大对象,From/To不足→FullGC
  - 避免新生代设置过大
    - 旧生代变小，频繁FullGC
    - 新生代变大，YoungGC更耗时

代大小调优策略

- 避免Eden过小或过大
  - 避免Eden设置过小
    - 频繁YoungGC
  - 避免Eden设置过大
    - From/To，频繁FullGC
  - 建议
    - Eden:From:To=8:1:1
- 合理设置新生代存活周期
  - 避免设置过小
    - 频繁FullGC
  - 避免设置过大
    - From/To占满也会Full GC
  - 默认值15

程序优化策略

- 文件
  - 异步写
  - 批量读
  - 限流
  - 限制文件大小
- 网络IO
  - 释放不必要引用
  - 使用对象缓冲池
  - 缓存失效算法：LRU、FIFO
  - 使用SoftReference (内存不够回收)
    WeakReference(FullGC回收)

建议

- 64位机,配置不同于32位
- 多核
- 内存能大些
- XMX=XMS，MaxPermSize=MinPermSize，减轻伸缩堆大小
- 调试，如-XX:+PrintClassHistogram-XX:+PrintGCDetails- XX:+PrintGCTimeStamps-XX:+PrintHeapAtGC-Xloggc:log/gc.log
- 若用了缓存，旧生代应该大一些，HashMap不应该无限制长，建议采用LRU算法的Map做缓存，LRUMap的最大长度也要根据实际情况设定
- 永久代会逐渐变满，所以隔三差五重起java服务器是必要的

展望未来

- JDK8
- HotSpot+JRockit
- 取消Perm Gen
- 一种新的内存空间诞生-Metaspace

问题答疑